SETN2016

AI-IoT2016

A Semi-Automatic Approach for Semantic IoT Service Composition

Grigorios Tzortzis and Evaggelos Spyrou
Institute of Informatics and Telecommunications
NCSR “Demokritos”, Greece

SYNAISTHISI

Problem Definition (1)

Internet of Things (IoT)

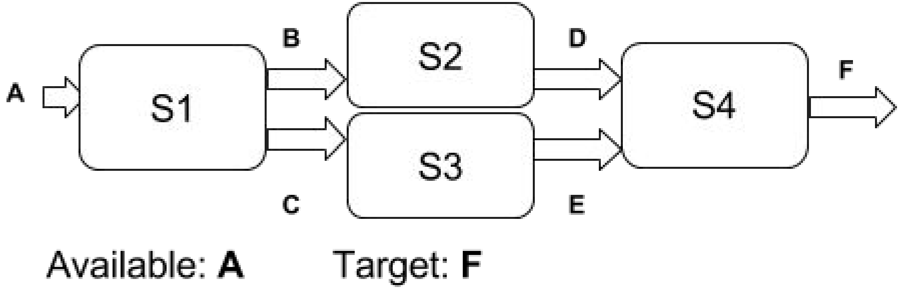
- Integration of everyday physical objects with the world wide web
- IoT adopts a service-oriented architecture (SoA), where all “things” are exposed as (semantic) web services

Service Composition in IoT

- Combine the available services in an IoT ecosystem to construct a new, composite service that fulfils some desired functionality
- Discover appropriate services and interconnect them
- Ensure that all services are invokable

Problem Definition (2)

Service composition is primarily about matching service outputs (and effects) to service inputs (and preconditions)



Approaches to service composition

- Manual vs Semi-automatic vs Automatic
- Syntactic vs Semantic

Necessary Tools

- Ontology
  - Semantic annotations for services
  - We propose a smart meeting room ontology
- IoT-ready platform
  - Interconnection and coordination of vast number of heterogeneous devices
  - Devices exposed as services
  - Ontology support and reasoning over ontologies
  - We use the SYNAISTHISI platform[1] developed at IIT, NCSR “Demokritos”

[1] G. Pierris et al., *SYNAISTHISI: An Enabling Platform for the Current Internet of Things Ecosystem*, PCI, 2015

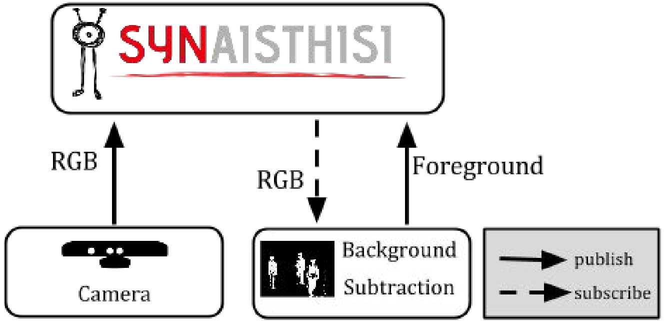
The SYNAISTHISI platform (1)

- Available services are registered into a service registry, implemented by an RDF triplestore

- They follow the IoT paradigm and are divided into:
  - **S**-type services corresponding to sensors that sense the physical world
  - **P**-type services corresponding to processors (algorithms) that process the measurements of the S-type services and/or the processed results of other P-type services
  - **A**-type services corresponding to actuators that are used for the actuation of devices/signals based on the acquired results.

The SYNAISTHISI platform (2)

- Services exchange information with messages via the MoM
- Information is shared by “publishing” through specific topics
- Services that need to use information, “subscribe” to the appropriate topics



Smart Meeting Room Ontology[1]

Models the SPA services of smart meeting rooms

- Domain-specific
- High-level and low-level concepts
- Enhance service discovery and composition
- Reuses existing ontologies — IoT-A, SSN, QU, QUDT
- Integrated into the SYNAISTHISI platform

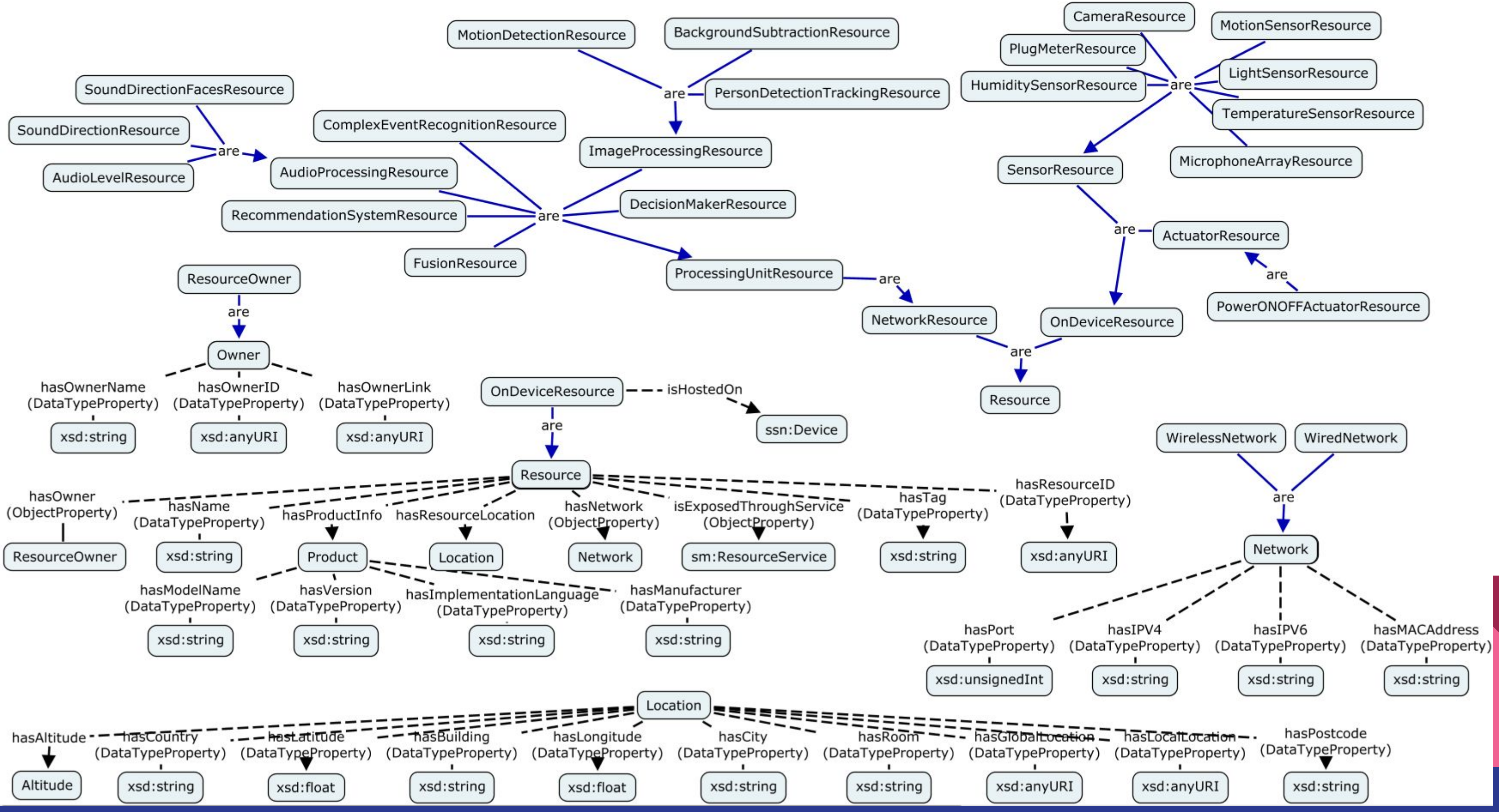
Statistics

- ~200 classes
- ~50 Datatype properties
- ~50 Object properties

[1] Preliminary version: C. Akasiadis et al., *Developing Complex Services in an IoT Ecosystem*, WF-IoT, 2015

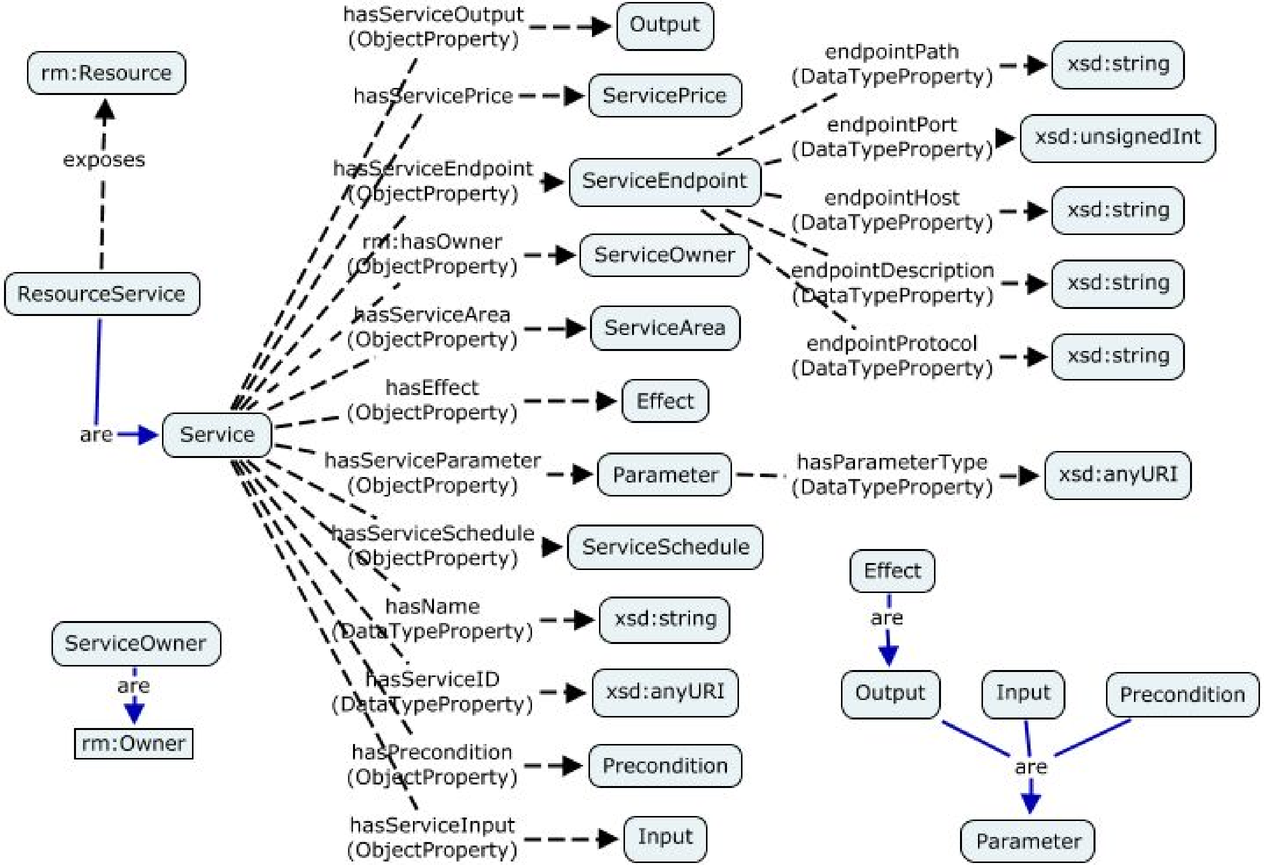
Resource Model (excerpt)

Describes the characteritics of the device hidden behind a service



Service Model (excerpt)

Describes the characteristics of the service exposing the device


rm:Resource
exposes
ResourceService
are
Service
ServiceOwner
are
rm:Owner
hasServiceOutput (ObjectProperty)
Output
hasServicePrice
ServicePrice
hasServiceEndpoint (ObjectProperty)
ServiceEndpoint
rm:hasOwner (ObjectProperty)
ServiceOwner
hasServiceArea (ObjectProperty)
ServiceArea
hasEffect (ObjectProperty)
Effect
hasServiceParameter (ObjectProperty)
Parameter
hasServiceSchedule (ObjectProperty)
ServiceSchedule
hasName (DataTypeProperty)
xsd:string
hasServiceID (DataTypeProperty)
xsd:anyURI
hasPrecondition (ObjectProperty)
Precondition
hasServiceInput (ObjectProperty)
Input
endpointPath (DataTypeProperty)
xsd:string
endpointPort (DataTypeProperty)
xsd:unsignedInt
endpointHost (DataTypeProperty)
xsd:string
endpointDescription (DataTypeProperty)
xsd:string
endpointProtocol (DataTypeProperty)
xsd:string
hasParameterType (DataTypeProperty)
xsd:anyURI
Effect
are
Output
Input
Precondition
are
Parameter

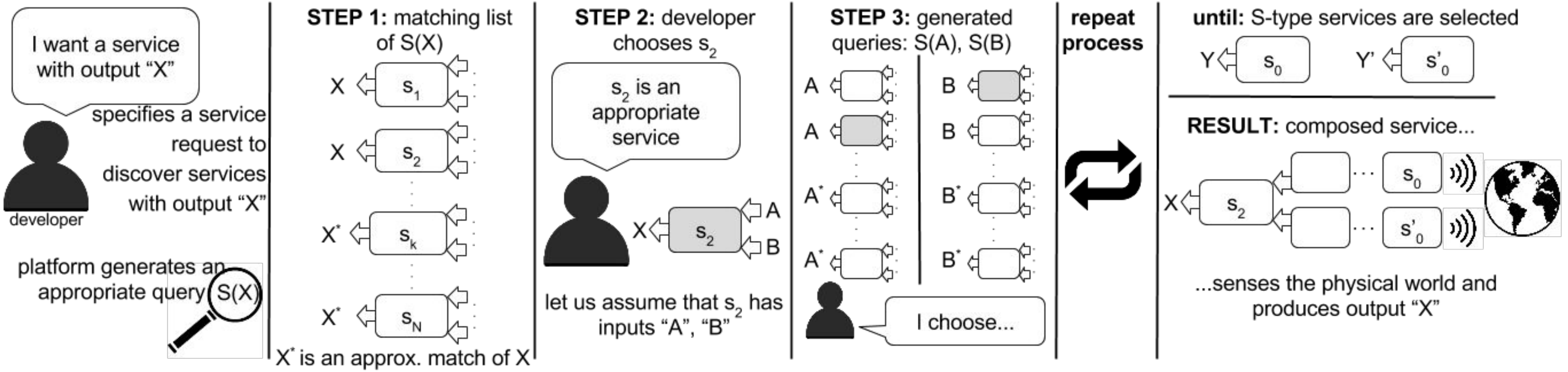
Service Composition

We propose a semi-automatic approach for SPA service composition as part of the SYNAISTHISI platform

Main features

- Utilizes semantics of the smart meeting room ontology
- Minimum human intervention
  - The platform guides the developer in building a composite service
  - Service discovery and interconnection is the responsibility of the platform
- Based on matching services’ outputs to inputs
  - Preconditions are ignored, effects are treated as special type of output of A-type services

Service Composition Algorithm



I want a service with output "X"

specifies a service request to discover services with output "X"

developer

platform generates an appropriate query S(X)

STEP 1: matching list of S(X)

X^* is an approx. match of X

STEP 2: developer chooses s_2

s_2 is an appropriate service

let us assume that s_2 has inputs "A", "B"

STEP 3: generated queries: S(A), S(B)

I choose...

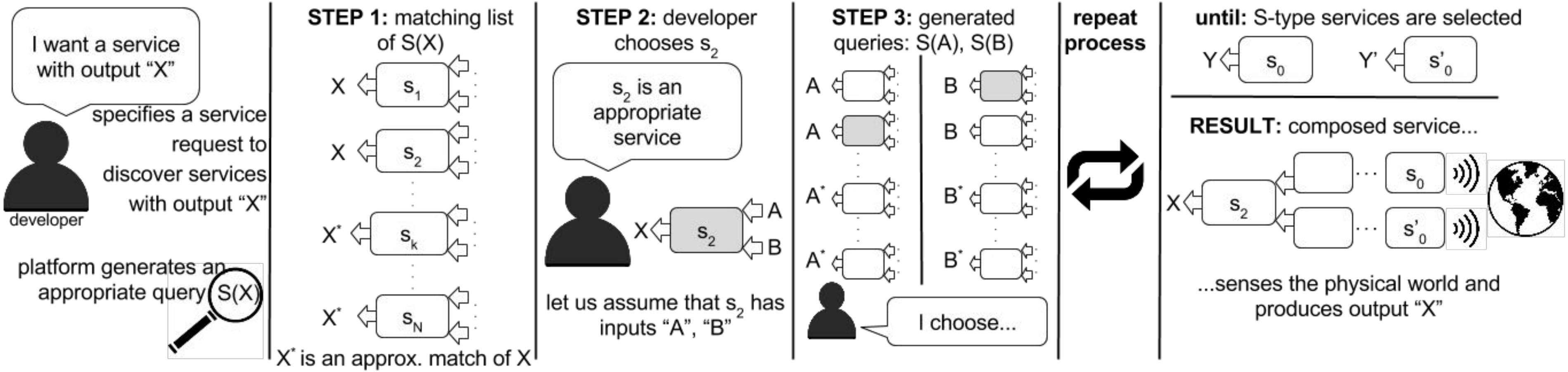
repeat process

until: S-type services are selected

RESULT: composed service...

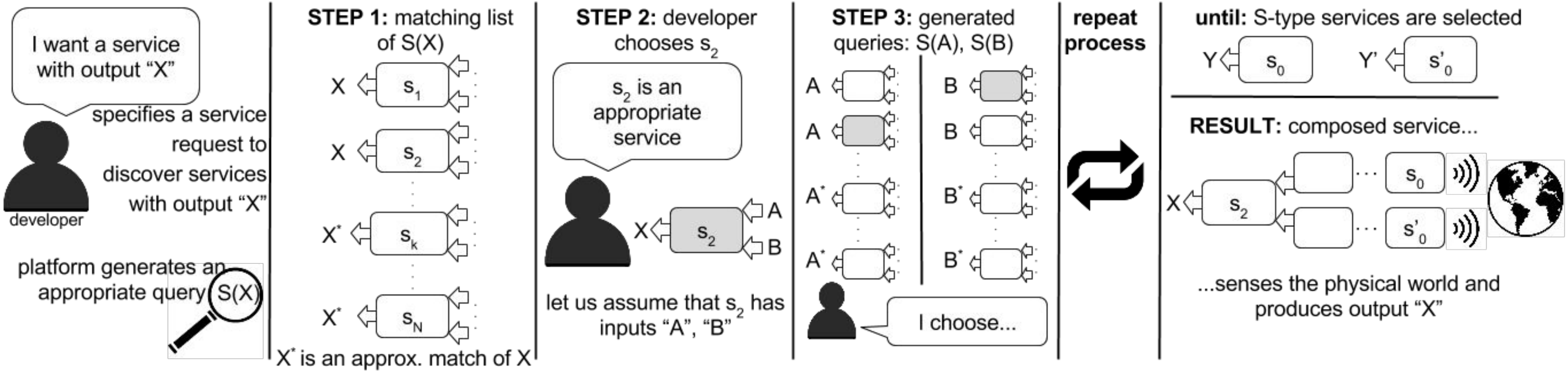
...senses the physical world and produces output "X"

Service Composition Algorithm


I want a service with output "X"
developer
specifies a service request to discover services with output "X"
platform generates an appropriate query S(X)
STEP 1: matching list of S(X)
X s1
X s2
X* sk
X* sN
X* is an approx. match of X
STEP 2: developer chooses s2
s2 is an appropriate service
X s2 A B
let us assume that s2 has inputs "A", "B"
STEP 3: generated queries: S(A), S(B)
A A A* A*
B B B* B*
I choose...
repeat process
until: S-type services are selected
Y s0 Y' s'0
RESULT: composed service...
X s2 ... s0 ... s'0
...senses the physical world and produces output "X"

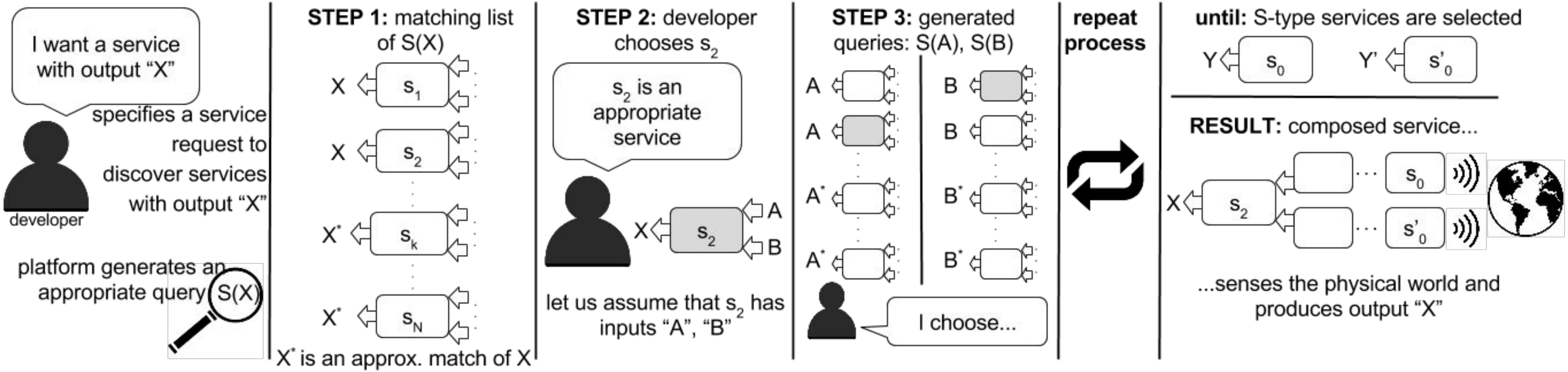
- The service request contains the desired outputs of the composite service
  - Outputs are declared using a suitable concept from an ontology

Service Composition Algorithm


I want a service with output "X"
specifies a service request to discover services with output "X"
developer
platform generates an appropriate query
S(X)
STEP 1: matching list of S(X)
X
s1
X
s2
X*
sk
X*
sN
X* is an approx. match of X
STEP 2: developer chooses s2
s2 is an appropriate service
X
s2
A
B
let us assume that s2 has inputs "A", "B"
STEP 3: generated queries: S(A), S(B)
A
A
A*
A*
B
B
B*
B*
I choose...
repeat process
until: S-type services are selected
Y
s0
Y'
s'0
RESULT: composed service...
X
s2
s0
s'0
...senses the physical world and produces output "X"

- The service request contains the desired outputs of the composite service
    - Outputs are declared using a suitable concept from an ontology
- A matching list contains services whose output matches a particular input of another service
    - An independent service discovery is launched to populate the matching list
    - Utilization of semantics in finding matches (explained later)
    - The developer must choose a service from the matching list presented to him

Service Composition Algorithm


I want a service with output "X"
specifies a service request to discover services with output "X"
developer
platform generates an appropriate query
S(X)
STEP 1: matching list of S(X)
X
s_1
X
s_2
X^*
s_k
X^*
s_N
X^* is an approx. match of X
STEP 2: developer chooses s_2
s_2 is an appropriate service
X
s_2
A
B
let us assume that s_2 has inputs "A", "B"
STEP 3: generated queries: S(A), S(B)
A
A
A^*
A^*
B
B
B^*
B^*
I choose...
repeat process
until: S-type services are selected
Y
s_0
Y'
s'_0
RESULT: composed service...
X
s_2
s_0
s'_0
...senses the physical world and produces output "X"

- The service request contains the desired outputs of the composite service
  - Outputs are declared using a suitable concept from an ontology
- A matching list contains services whose output matches a particular input of another service
  - An independent service discovery is launched to populate the matching list
  - Utilization of semantics in finding matches (explained later)
  - The developer must choose a service from the matching list presented to him
- If non-empty matching lists are found for all inputs of a service, the service is invokable
  - S-type services can be readily invoked when selected → Service discovery is unnecessary

Semantic Relaxation

Exploit semantic hierarchical relationships to decide if output concept `A` matches input concept `B`

- `exact(A,B)` → Same URI or OWL equivalent
- `plugin(A,B)` → `A` is subsumed by `B`
- `subsume(A,B)` → `A` subsumes `B`

} Approximate match

`exact < plugin < subsume` in terms of semantic relaxation degree

Advantages of the Service Composition Approach

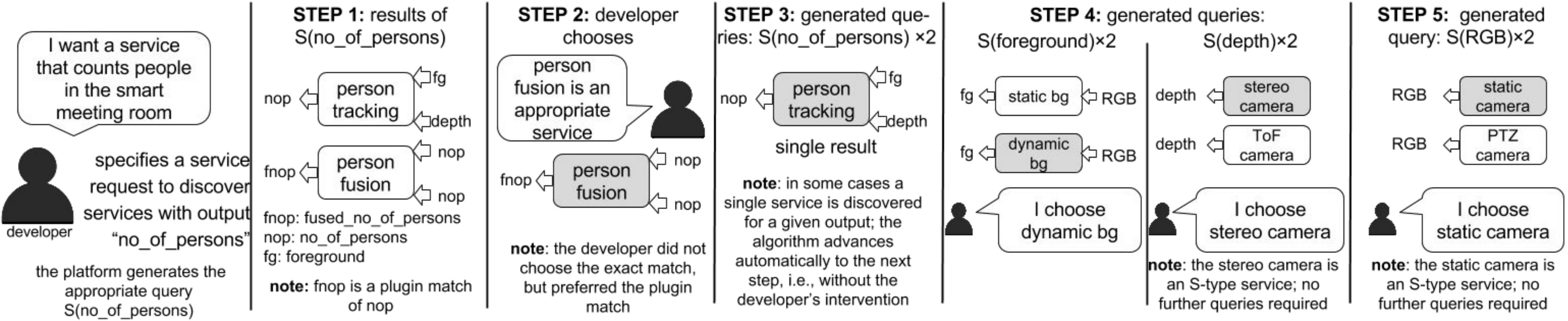
- Guarantees that the composite service:
  - Satisfies the service request
  - All its services can be invoked
- Semantic relaxation
  - Avoid syntactic barriers
  - Permit approximate solutions when exact ones do not exist
- Service discovery and interconnection is the responsibility of the platform
- The developer only defines a service request and selects services from platform-generated matching lists
  - Even experienced users can perform service composition

Use Case: Creating a People Counting Service (1)

- One of the pilots of the SYNAISTHISI project was a smart meeting room
- Among the goals was the minimization of user discomfort, environmental impact and monetary costs
- To achieve these goals, an estimation of the number of people present within the room was necessary
- Since cameras were installed, a computer vision approach was followed
- Several services were developed to support the functionalities of the smart meeting room

Use Case: Creating a People Counting Service (2)

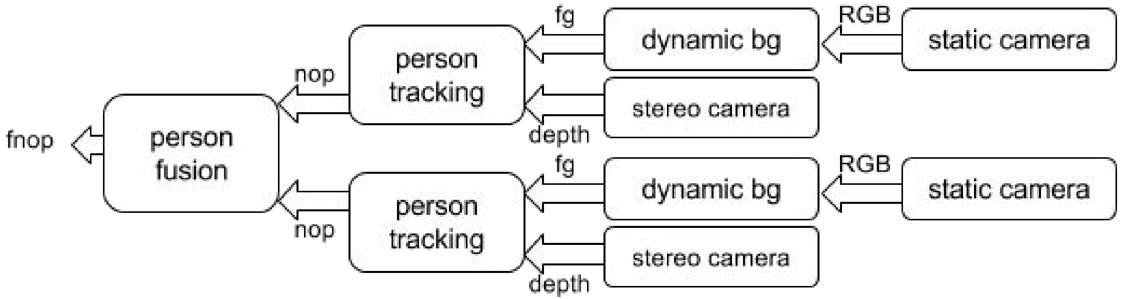
The complex people counting service may be composed using simpler services


I want a service that counts people in the smart meeting room
developer
specifies a service request to discover services with output "no_of_persons"
the platform generates the appropriate query S(no_of_persons)
STEP 1: results of S(no_of_persons)
nop
person tracking
fg
depth
fnop
person fusion
nop
nop
fnop: fused_no_of_persons
nop: no_of_persons
fg: foreground
note: fnop is a plugin match of nop
STEP 2: developer chooses
person fusion is an appropriate service
fnop
person fusion
nop
nop
note: the developer did not choose the exact match, but preferred the plugin match
STEP 3: generated queries: S(no_of_persons) ×2
nop
person tracking
fg
depth
single result
note: in some cases a single service is discovered for a given output; the algorithm advances automatically to the next step, i.e., without the developer's intervention
STEP 4: generated queries:
S(foreground)×2
fg
static bg
RGB
fg
dynamic bg
RGB
I choose dynamic bg
S(depth)×2
depth
stereo camera
depth
ToF camera
I choose stereo camera
note: the stereo camera is an S-type service; no further queries required
STEP 5: generated query: S(RGB)×2
RGB
static camera
RGB
PTZ camera
I choose static camera
note: the static camera is an S-type service; no further queries required

The developer should have a basic knowledge in the field of computer vision

Use Case: Creating a People Counting Service (3)

The resulting composite service:



More details and evaluation of this algorithm may be found in:

D. Sgouropoulos, E. Spyrou, G. Siantikos and T. Giannakopoulos, *Counting and Tracking People in a Smart Room: an IoT Approach*, SMAP, 2015.

Open Issues and Future Work (1)

- Use lightweight standards to annotate services and their IOPEs
  - SAWSDL, hRest
  - More native to services

- Pursue semantically-aware automatic service composition
  - Graph-based
  - AI planning-based
  - Appropriate for end-users that are not developers

Open Issues and Future Work (2)

- Service composition should consider:
  - Functional requirements
  - Non-functional requirements e.g. location, reputation, QoS
  - User preferences

- Service marketplace that supports the full cycle of producing, delivering and trading a service
  - Exploit service composition to deliver complex applications to end-users

Thank you!

Questions?